

AD-A093 874

MICHIGAN UNIV ANN ARBOR SYSTEMS ENGINEERING LAB
A BLOCK-ORIENTED EQUATION SOLVER FOR THE CRAY-1.(U)
DEC 80 D A CALAHAN
SAFOSR-80-0158

F/G 9/2

UNCLASSIFIED

AFOSR-Tr-80-1375

NL

1 of 1
8/18/84



END
DATE
FILMED
2-81
DTIC

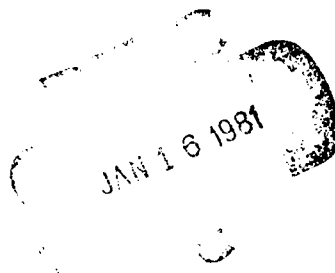
LEVEL

(4)

AD A093874

A Block-Oriented Equation Solver for the CRAY-1

D. A. Calahan



December 1, 1980

Sponsored by
Directorate of Mathematical and Information Sciences,
Air Force Office of Scientific Research
under Grant [REDACTED]

AFOSR-80-136

SYSTEMS ENGINEERING LABORATORY

DDC FILE COPY



Approved for public release;
distribution unlimited

81 1

16 022

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM	
1. REPORT NUMBER (18) AFOSR-TR-80-1375	2. GOVT ACCESSION NO. AD-A093874	3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) (6) A BLOCK-ORIENTED EQUATION SOLVER FOR THE CRAY-1.	5. TYPE OF REPORT & PERIOD COVERED Interim report	6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s) (10) D. A./Calahan	8. CONTRACT OR GRANT NUMBER (if any) (1E) AFOSR-80-0158	9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Michigan Dept. of Electrical & Computer Engring. Ann Arbor, MI, 48109	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS (16) 2304/A3 611021
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research Bolling AFB, Washington, DC, 20332	12. REPORT DATE (11) (NM) Dec 1980	13. NUMBER OF PAGES 31	14. SECURITY CLASS. of this report (12) 38 UNCLASSIFIED
15. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) (14) SEL-136	16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in block 20, if different from Report)			
18. SUPPLEMENTARY NOTES			
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Sparse matrices Vector processing Parallel computation			
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) (1) This report describes algorithms, applications, and use of a general block-oriented sparse equation solver for a memory-hierarchical, functionally current vector processor, the CRAY-1.			

(4)

A Block-Oriented Equation Solver
for the CRAY-1

D. A. Calahan

Systems Engineering Laboratory
University of Michigan
Ann Arbor, Michigan 48109
December 1, 1980
SEL Report #136

Sponsored by the
Directorate of Mathematical and Information Sciences
Air Force Office of Scientific Research
Under Grant [REDACTED]

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (AFSC)
NOTICE OF TRANSMITTAL TO DDC
This report has been reviewed and is
approved for public release IAW AFR 190-12 (7b).
Distribution is unlimited.
A. D. BLOOM
Technical Information Officer

Abstract

This report describes algorithms, applications, and use of a general block-oriented sparse equation solver for a memory-hierarchical, functionally concurrent vector processor, the CRAY-1.

Acknowledgment

The usefulness of a CRAY-1 clock-level simulator written by D. A. Orbits in developing this high performance code is acknowledged.

Availability

These programs are available on a user-supplied 9 track tape. Specify desired density, labelling, and blocking.

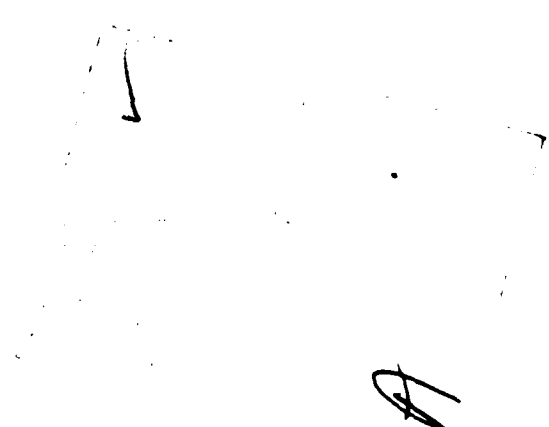


Table of Contents

	PAGE
I. INTRODUCTION	
A. Sparse Equation Solvers.	1
B. Block Equation Solvers	2
1. Kernels of sparse solvers.	2
2. Block solver characteristics	4
II. ALGORITHM ORGANIZATION	
A. Introduction	6
B. Global Blocking.	7
C. Local Blocking	7
D. Numeric Storage Considerations	11
E. Solution of Band Equations	12
III. TIMING MODELS AND PERFORMANCE EVALUATION	
A. Introduction	15
B. Example: Block Tridiagonal Matrices	15
C. Example: Banded Matrices.	15
D. Other Timing Considerations.	18
1. Efficiency	18
2. Bounds on Execution Rate	19
E. Precise Timing Simulation.	20
IV. PROGRAM DESCRIPTION AND USER INFORMATION	
A. Introduction	22
B. A general Driver Program	22
1. High Level Descriptors	22
2. Equation Formulation	25
3. A Fortran Simulator.	25
C. Generation of Block Descriptors.	25
Conclusions	29
References.	30

PREFACE

Because the body of the report considers detailed algorithmic issues, this preface is intended to present the principal rationale which motivated development of this software, and which would justify the reader to consider its adoption.

Heretofore, sparse matrix methods have been used only for the solution of somewhat unusually-structured matrices; however, the specialized structure of hierarchically-organized vector processors such as the CRAY-1 suggest important new applications of this methodology.

For example, the solution of a small full matrix through Fortran will execute at an asymptotic rate one-third the maximum processing rate of the CRAY-1, due to the difficulty of managing the vector register cache memory through Fortran. This limitation can be corrected at present only by using assembly language or by unorthodox Fortran programming. Moreover, when the matrix size increases beyond 64, vector length limitations require that, to maintain high execution rates, the full matrix be artificially blocked and the matrix reduced a block at a time. Thus the user is threatened with unaccustomed programming and algorithmic detail for what may appear to be a routine linear algebra task.

The programming difficulty increases substantially when the matrix itself has a sparsity structure. For example, the maintenance of high processing rates at the bandedge of a band matrix -- partitioned for the above reason -- is a difficult task. For this reason, at the time of this writing, no high performance general band solver is known to exist. Matrix structural irregularities resulting from boundary conditions associated with solution of partial differential equations are even more challenging. As a result, the discipline-oriented researcher is initially inclined to accept the above-mentioned 3:1 degradation obtainable from a vectorized Fortran code.

By exploiting traditional sparse matrix methodology, this partitioning and high performance blocked solution can be performed automatically or at least under the high level control of an enlightened user. In this software, two levels of control of the solution process are provided.

(a) Certain common types of matrix structures (full, banded, block tri-diagonal) can be described by several parameters (matrix size, bandwidth, etc.); a partitioned matrix description is automatically formed and the normal calls to numerical factorization and substitution codes carried out by the user.

(b) At the lowest level, two levels of block descriptors can be provided by the user who has a matrix sparsity structure not provided for above.

Because reduced execution time is the primary motivation for utilization of this unfamiliar software, it is important to give the reader/user a means of estimating this timing. Three levels of performance prediction are provided.

(1) If block sizes are of a constant or a known minimum size, bounds on the execution rates can be given independently of the matrix sparsity structure (Table 5).

(2) Tables of solution times for common matrix structures are also given in this report.

(3) A precise timing model of the software package has been devised using a CRAY-1 timing simulator and captured in a Fortran program. The result is that a Fortran version of the software exists that produces, besides a numerical solution, a precise timing estimate of this numerical solution on the CRAY-1. This is intended to be useful to determine, on a local scalar machine, the speedups achievable with the CRAY-1 software, given either of two levels of problem descriptions above.

I. INTRODUCTION

A. Sparse Equation Solvers

General sparse equation solvers are complicated software packages which solve directly (contrast iteratively) simultaneous linear equations having an arbitrary off-pivot sparsity structure [1] - [2]. This structure is described either by a linked list [3] or a bit map [4]. Such solvers have been used for more than a decade as kernels in (1) the implicit solution of algebraic and ordinary differential equations associated with lumped physical systems such as electronic circuits, electrical power systems, and 3-D mechanisms, and (2) the solution of discretized partial differential equations (PDE's) such as arise in oil reservoir simulation.

This report examines the characteristics and applications of such solvers implemented on a memory-hierarchical vector processor, the CRAY-1. Because the architecture is fixed, a "bottom-up" approach is used, wherein the performance of appropriate numerical kernels on the CRAY-1 is studied first. These kernels are shown to have a strong preference for block matrix structure. This single fact profoundly affects the development of the solver and its applications, discussed in the remainder of the paper.

Because this study is directed toward a specific processor, it is first well to summarize three general issues which implementation of such a code on any vector processor are likely to involve.

(a) Algorithmically, vectors arise from either local matrix density (coupling of variables) or from symmetries in globally decoupled parts of the problem structure. The former will be considered in this report; the latter is discussed in [5].

(b) The coding of such a solver will profoundly affect its efficiency; this coding in turn is intimately related to the data flow of the processor, in part due to the linked list or bit processing. This implies a need for coding in a low level language and a consequent high degree of machine dependency. The CRAY-1 was chosen for this study due to its superior scalar and short vector performance. Linked lists rather than bit maps were adopted because of the CRAY-1's

limited bit-controllable vector operations.

(c) The performance evaluation involves construction of a timing model so that the price paid for using a vector processor and a general equation-solving code can be evaluated by a potential user. In this study, the general solver will be shown to execute faster than codes written for specific matrix structures and conventionally coded in CAL (Cray Assembly Language).

B. Block Equation Solvers

1. Kernels of Sparse Solvers.

The proposed general sparse equation solver operates on linked list descriptions of the matrix structure. Because vector operations utilizing linked lists (termed "gathering" and "scattering") commonly proceed at 1/5 to 1/10 the speed of linearly-indexed array computation, it is important for efficient vector operation that the list not point to a single non-zero element. Assuming significant local matrix coupling, the list should point to either

- (a) a dense segment of a row or column, or, more generally,
- (b) a dense block

of the matrix. With sufficient numeric computation resulting from this density, the index list processing can be overlapped or at least overwhelmed by the numeric computation.

Consider the factorization of a matrix A into upper and lower triangular forms U and L respectively. The numeric kernel is commonly of the form

$$D_i + D_i - B_i C_i \quad (1)$$

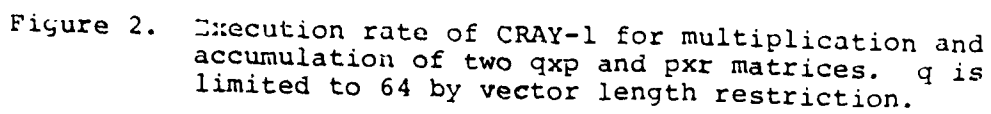
where D_i , B_i , and C_i are submatrices from the overall sparse systems matrix. The accumulation of the column segments of (a) will be termed a line-vector operation of the form

$$A_{i:j,k} + A_{i:j,k} - u_{l,k} L_{i:j,l} \quad (2)$$

Diagram illustrating the accumulation of blocks:

- A block labeled $U_{i:j,k:l}$ is shown.
- An arrow points from this block to a larger block labeled $\Lambda_{p:q,k:l}$.
- The larger block $\Lambda_{p:q,k:l}$ is composed of several smaller blocks, one of which is labeled $L_{p:q,i:j}$.

Figure 1. Accumulation models



where $A_{i:j,k}$ and $L_{i:j,\ell}$ are dense column segments of the matrix A and the triangular factor L , extending from row i to row j , and $u_{\ell,k}$ is an element of U (Figure 1(a)). A block-vector operation can be written (Figure 1(b)).

$$A_{i:j,p:q} \leftarrow A_{k:j,p:q} - L_{i:j,k:\ell} U_{k:\ell,p:q} \quad (3)$$

with similar subscript notation to indicate initial and final row and column indices.

An approximate timing model for the line-vector accumulation loop on the CRAY-1 has been found by simulation to be

$$\text{MFLOPS} = 53.3 \left(\frac{1}{1 + 31.3/\bar{\ell}} \right) \quad (4)$$

where $\bar{\ell}$ is the average length of a dense column segment encountered during the accumulation. This model, valid when all vectors are longer than 14, achieves a maximum rate of 35.8 for $\bar{\ell} = 64$, the maximum vector length on the CRAY-1.

The mathematical model for the vector block accumulation of (3) is given in Figure 2 as a function of block dimensions. The asymptotic rate of this kernel is 151 MFLOPS for $q = 64$ (the vector length) and p and q large.

The ratio of asymptotic execution rates ($151/35.8 = 4.2$) can be traced directly to the memory bandwidth required to implement the line-vector accumulation of (2), where an element of $A_{i:j,k}$ and $L_{k:j,\ell}$ must be loaded and $A_{i:j,k}$ stored for each and multiply. With a single memory path transferring one word in 12.5 nsec, the low asymptotic rate of $2(80 \times 10^6)/3 = 53.3$ MFLOPS of (4) follows.

2. Block Solver Characteristics

The above preference for block processing is unfortunate in the sense that such blocking is not unique and little is known of "optimal" methods of blocking a general sparse matrix. The equation formulation and the "fill" production [6] produce considerable overlapping of block structures, often masking the original problem structure. An example blocked structure is shown in Figure 3. Thus, a block solver must either be given the block structure or await the development of

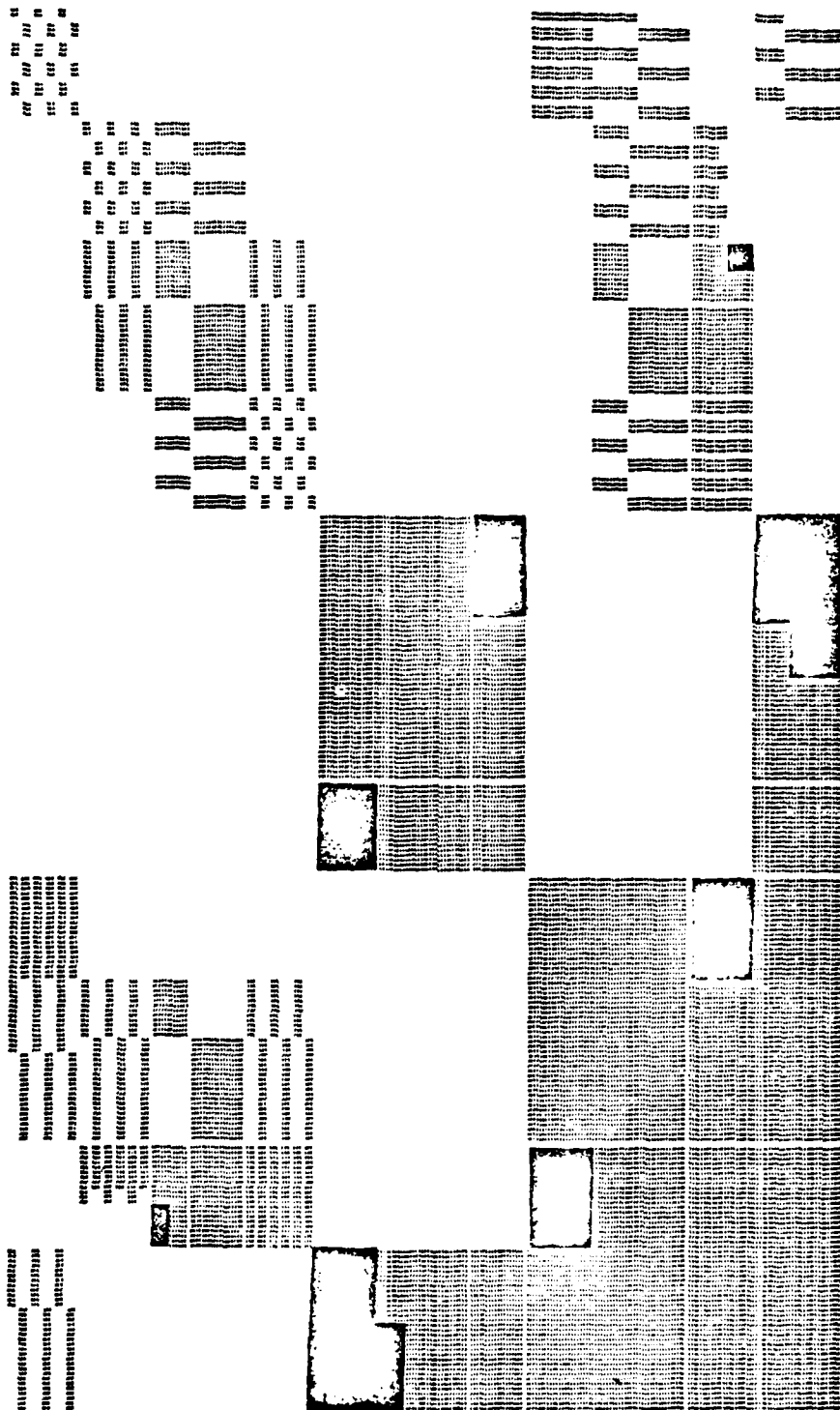


Figure 3. Block structure of dissected finite element matrix; boundary conditions account for some irregularities in block structure. Blackened regions indicate where non-zeros were added to conform to blocking rules.

blocking algorithms. This report considers only the development of the solver, in part because general blocking methods will undoubtedly be dependent on characteristics of the solver.

A general block-oriented equation solver for a vector processor, although conceptually derived from previous single-element processing codes, portends to have several new important applications.

(a) Common matrix structures are naturally blocked by local coupling and variables of nodes. The speed of vector processors encourage such coupling, especially to speed up the convergence of global iterative methods. For example, block relaxation methods are being considered to replace single-variable relaxation [7].

(b) A memory-hierarchical processor with a limited vector length requires the partitioning of large dense systems to reduce data flow between hierarchies -- the source of difficulties in the line-vector accumulation above. Although specialized partitioned codes can be developed for each general class of dense matrix (full, band, profile, block tridiagonal), the coding in a low level language for vector processors is sufficiently complicated that, at this writing, only partitioned full matrix codes have been developed for the CRAY-1 [8].

If one observes that partitioning is a symbolic rather than numeric process, it is reasonable to propose a solver which requires the user to construct only a set of block descriptors from the matrix structure. These descriptors would then guide the numeric solution. The descriptors must be adequate to describe common block structures and typical numeric storage schemes, but limited in number so that they do not seriously interfere with the numeric kernels, on which the processing speed of the solver depends.

II ALGORITHM ORGANIZATION

A. Introduction

The algorithm can be divided into two conceptual and organizational levels: (1) the global level, where general blocking rules are established and a general block-oriented solution algorithm is presented, and (2) the local level where sub-block structure forces tradeoffs between generality and speed.

B. Global Blocking

It is proposed that the blocking be performed on the LU map of the matrix since it is the structure of L and U rather than the structure of A which is important at each stage of the solution process. Further, it is proposed that the blocking be based on the size of diagonal blocks. In the general sparse case, these blocks are determined by scanning the matrix diagonal (in one direction) for the largest full, square blocks. This yields a unique diagonal blocking. Row and column strips are defined throughout the matrix based on these diagonal block partitions. Off-diagonal blocks are then defined only within the intersection of row and column strips. Figure 3 shows an example of this blocking.

With the blocks constrained to lie within such strips, it is possible to specify the block solution algorithm. In Figure 4, the factorization steps are given for a blocked matrix of the form

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1n_b} \\ A_{21} & A_{22} & & \\ \vdots & & \ddots & \\ A_{n_b 1} & & & A_{n_b n_b} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n_b} \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_{n_b} \end{bmatrix} \quad (5)$$

where the A_{ii} are square dense blocks and the sparsity of the A_{ij} ($i \neq j$) is as yet unconstrained.

C. Local Blocking

Define diagonal-based column and row strips (DBCS and DBRS, respectively) as those portions of column and row strips extending from each diagonal to the southern and eastern boundaries of the matrix, respectively. These strips contain all of the A_{ij} and A_{ii} blocks of Figure 4; it is their sparsities which are now of concern in the general sparse case.

An examination of the substitution step shows that any nonzero position in either blocks A_{ij} or A_{ki} will result in propagation or "raining" of nonzero positions to the eastern or southern boundary (respectively) of that block. This property is observed in the L

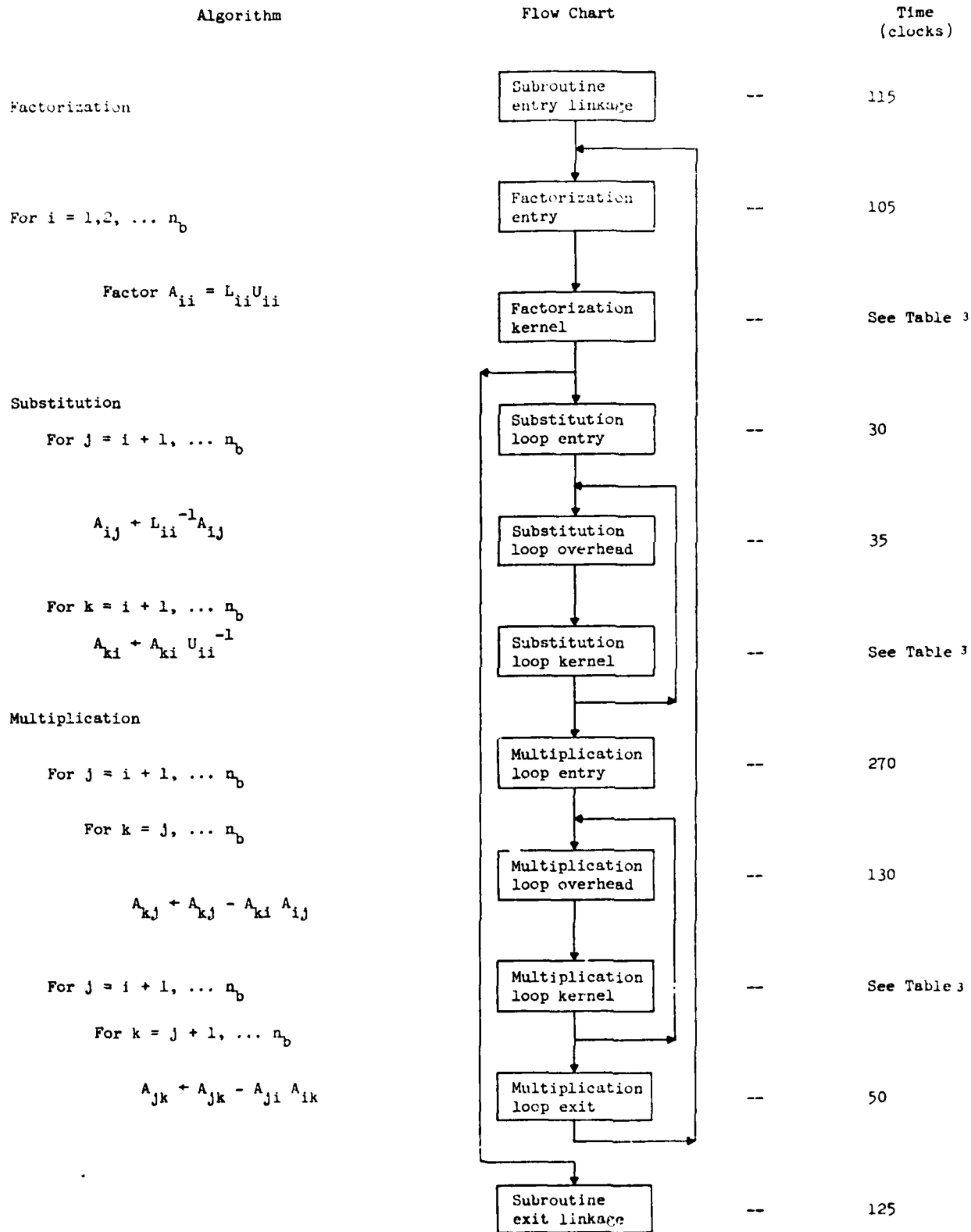


Figure 4. Algorithm and timing for factorization portion of sparse solver code.

and U map of Figure 3. This effect (a) fixes the contour along at least one of the sides of each block, and (b) requires the maximum length of any block along its DBCS or DBRS to be at its eastern or southern boundary, respectively (Figure 5).

Beyond this restriction, an arbitrary substructure of L and U is possible. To accommodate this generality in a solver requires block contour descriptors and, more importantly, slows the multiplication kernel of Figure 4. A compromise adopted for this study was to assume each sub-block extends across its DBRS or DBCS as shown in Figure 5; however, any number of blocks may lie within an intersection of a row and column strip. This assumption is valid for common types of block structures or when blocking dense matrices. However, when special equation ordering is used to avoid fill [9], nonzeros may have to be added to extend blocks across the DBRS and DBCS; these are shown as darkened areas in Figure 3.

With the above considerations in mind, the following are proposed as descriptors of DBRS blocks.

- (b1) the row position of the (1,1) block position;
- (b2) the number of columns in the block; the number of rows is fixed by the diagonal block size;
- (b3) the column strip number containing the block; this descriptor can be determined in a preprocessor step; it initiates the scan to locate A_{jk} in the multiplication step;
- (b4) the address of the (1,1) block position in the packed matrix numeric array; this permits an arbitrary numeric block storage;
- (b5) the address increment in the numeric array between the (i,j) and the (i,j+1) block position; this eliminates the need for contiguously-stored block columns.

These descriptors can completely describe the sparsity of the DBRS, under the above assumptions. A similar set of descriptors is used for the blocks in the DBCS with the terms "row" and "column" interchanged, except (b5) which remains the column address increment to represent the column storage of all blocks.

These descriptors are stored in a list in the processing order of the blocks. From the multiplication step of Figure 4, this order begins at the first DBCS, then the first DBRS, then the second DBCS,

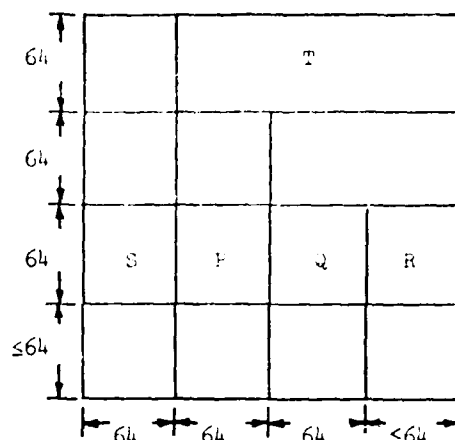
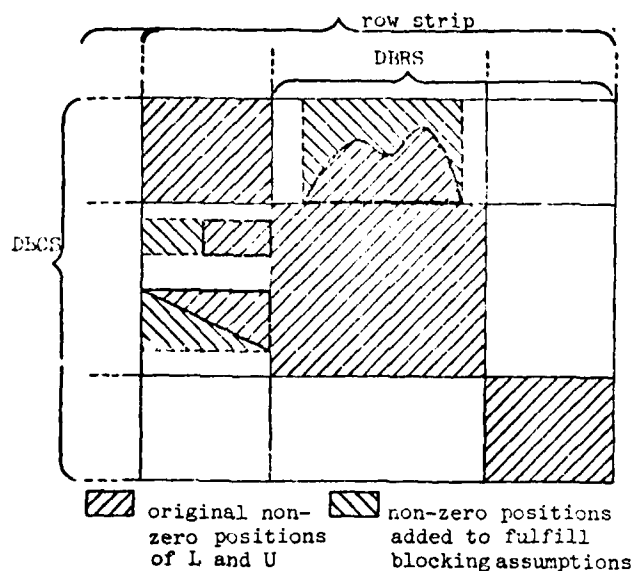
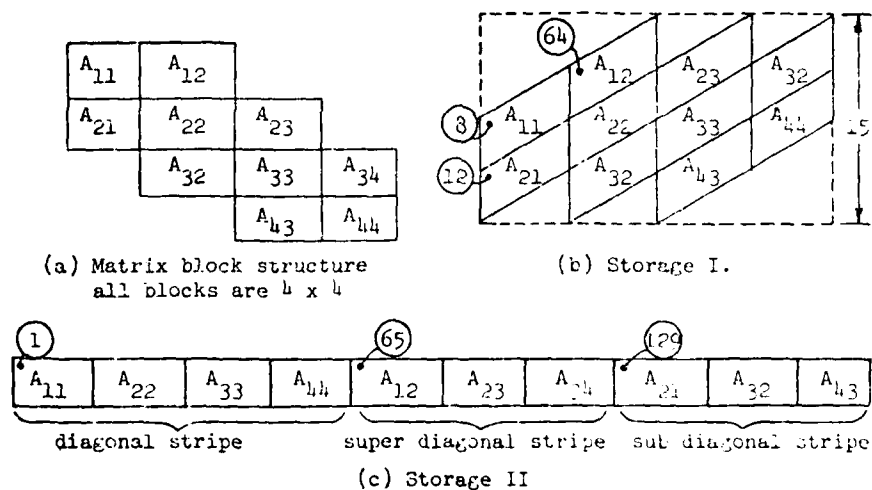


Figure 5. Local blocking assumptions

Figure 6. Simplified blocking of full matrix



Block	Storage	Symbolic descriptors			Numeric descriptors	
		b1	b2	b3	b4	b5
A ₁₁	I	1	4	1	8	14
	II	1	4	1	1	4
A ₂₁	I	5	4	2	12	14
	II	5	4	2	129	4
A ₁₂	I	5	4	2	64	14
	II	5	4	2	65	4

(d) Descriptors

Figure 7. Block tridiagonal matrix and two storage descriptions. Circled numbers refer to numeric array addresses.

etc., always beginning at the diagonal block. This is often termed Crout processing order.

D Numeric Storage Considerations

Whereas the symbolic pointers (b1-b3) are related to the order of block processing, the numeric pointers (b4-b5) are intended to allow an arbitrary block numeric storage. The result is that common compressed storage schemes can be accommodated; also, these pointers permit certain efficiencies to be taken in the blocking.

(a) Reducing the number of blocks. The restriction $q \leq 64$ in the numeric kernel of Figure 2 requires that row strips not exceed 64 in width. This is a universal restriction. Other of the previously mentioned global blocking restrictions were imposed to maintain the integrity of the numeric processing with arbitrary placement of blocks in the numeric storage. However, certain common storage schemes allow violation of these rules in the interest of efficiency. For example, a full matrix stored in column-ordered form in a numeric array could be partitioned as in Figure 6, with no column strip extending above the diagonal block. A single block multiply such as $S \cdot T$, if directed to be accumulated into P, would, in fact, be accumulated into blocks P-Q-R. One only has to insure that descriptor (b5) -- the address increment between columns of a block -- is the pseudo-row dimension of the numeric array representing the entire matrix. Of course, this freedom to violate a global blocking rule involves the risk of improper accumulation across several blocks or indeed into void areas of a sparse matrix. For example, no check is made whether the user has accounted for all the block fill in a sparse matrix in setting up the block descriptor list.

(b) Compressed format. In Figure 7 (b)-(c), several standard formats of block tridiagonal matrices are given. Storage I is similar to that of banded matrices, where the diagonals of the matrix are stored as rows of a two dimensional compressed array. This storage is achieved simply by reducing descriptor (b5) for all blocks to one less than the row dimension of the entire matrix array; a rectangular block contour then is skewed into a parallelogram as shown. Figure 7(d) shows that the numeric descriptors are different for the two storage schemes, but the symbolic pointers remain the same.

E. Solution of Band Equations

The numerical values of band matrices are normally stored in compressed form: each matrix diagonal is mapped into either a row or column of the compressed form. It can be shown that the former case (only) is compatible with the blocking scheme which has been proposed. The mapping compresses a band matrix of size $n_{bd} \times n_{bd}$ and half band-width n_{hb} into a storage of size $n_{hb} \times n_{bd}$. An example is shown in Figure 9(a)-(b).

The new problem introduced by the partitioning of a general band matrix occurs at the bandedge in the column and row strips. To process these partitions, an upper triangular block model is provided to reduce the bandedge of the column strip. A lower triangular block model serves the same purpose in a row strip. An additional descriptor $b6$ identifies such blocks in the list: $b6 = 0$ identifies an above-defined rectangular block; $b6 = 1$ identifies a bandedge block. Further descriptors are avoided by requiring that each block start precisely at the bandedge, with the first position in a triangular block on the first column/row of the column/row strip. A partial block, which occurs primarily at the top and bottom of the band edge, is then routinely described by $b2$, previously defined as the block length (Figure 8).

A band matrix is illustrated in Figures 9(a) and 9(b), the latter representing the compressed numerical storage. The six block descriptors of selected blocks of the matrix are given in Table 1. Among the noteworthy points raised by this example are the following.

- (a) Blocks $A_{i,i+1}$ extend from the i th diagonal to the band edge, regardless of the matrix size. This is similar to the full matrix case of Figure 6.
- (b) Blocks illustrated by $A_{41}^{(a)}$, $A_{52}^{(a)}$, and $A_{63}^{(a)}$ are reduced in size to extend to just short of the band edge, thus accommodating the triangular models of Figure 8.
- (c) Blocks such as $A_{41}^{(b)}$ overlap two row strips. However, the contiguous numerical storage of adjacent blocks assures that accumulation of products such as $A_{41}^{(b)} A_{12}^{(b)}$ into A_{42} will result in proper accumulations into adjacent blocks as well.

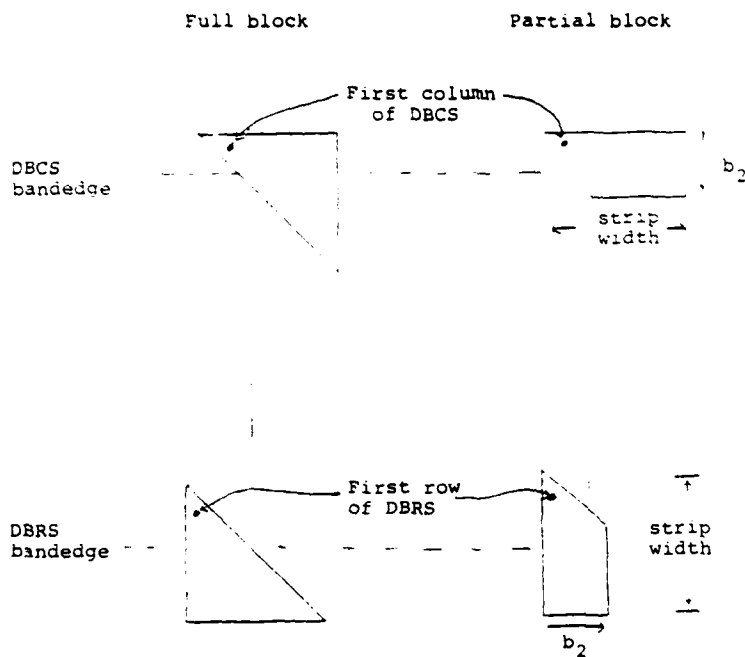
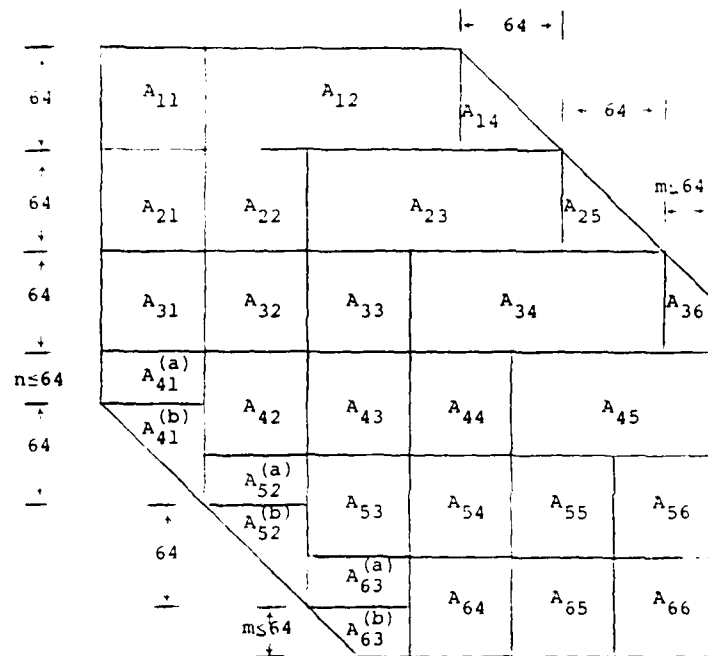


Figure 8. Models of bandedge partitions

Block	Symbolic Descriptors	Numeric Descriptors	Bandedge Descriptors
A	b1 b2 b3	b4 b5	b6
A_{11}	1 64 1	201 409	0
$A_{41}^{(a)}$	193 8 4	393 409	0
$A_{41}^{(b)}$	201 64 4	401 409	1
A_{14}	201 64 4	82001 409	1

Table 1. Selected block descriptors for compressed band matrix for $n_{hb} = 200$ ($n = 8$ in Fig. 9) with array row dimension 410.

(a) Matrix



(b) Compressed storage

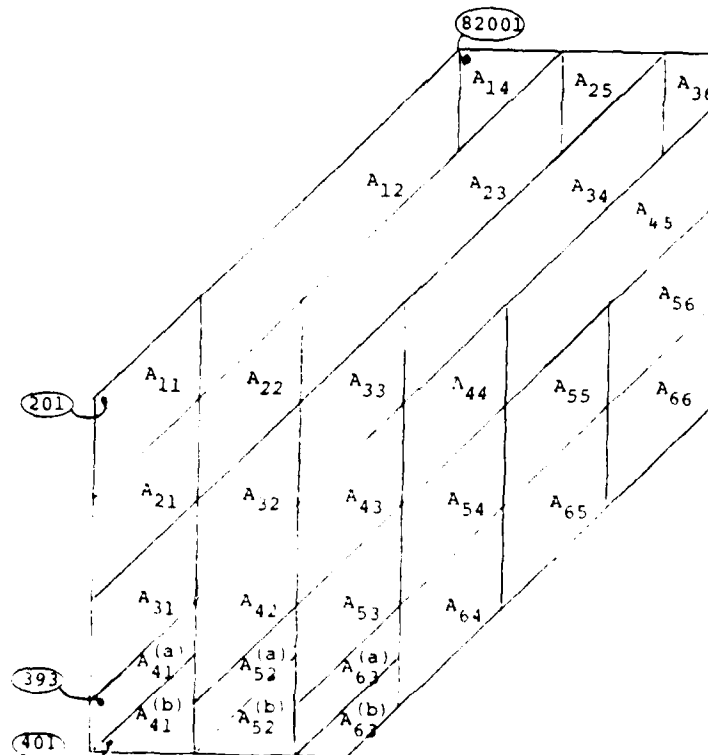


Figure 9. Partitioned band matrix and storage

III. TIMING MODELS AND PERFORMANCE EVALUATION

A. Introduction

Figure 4 depicts the major loops of the factorization program and yields, with Table 3, the corresponding timings of the overhead and kernel computations. These were developed using a timing simulator [10].

From these timings, it is possible to develop expressions for the solution time and the execution rate for any particular matrix structure.

B. Example: Block Tridiagonal Matrices

Vectorization of the solution of a single tridiagonal system on the CRAY-1 involves use of unconventional algorithms and, to achieve high execution rates, rather intricate assembly coding. Even so, the highest effective execution rate appears to be in the range of 20 MFLOPS [13]. In contrast, the solution of a coupled (or blocked) tridiagonal system can be vectorized simply by exploiting the density inherent in the coupling blocks.

For example, consider the solution of a block tridiagonal matrix with the general solver. The timing is determined from Figure 4 to be, for n_b nxn blocks,

$$T_F = 345 + (n_b - 1)(620 + T_f + T_s + T_m) + T_f$$

where T_f , T_s , and T_m are defined in Table 3. For a large number of blocks, the execution rate becomes simply (Table 4)

$$\text{MFLOPS} = \frac{(\text{arithmetic operations})}{T_F \times 10^6} = \frac{80(28n^3 - 9n^2 + 5n)}{6(620 + T_f + T_s + T_m)}$$

C. Example: Band Matrices

Table 2 gives the comparative performance of the sparse solver and a highly-tuned unsymmetric band solver [14][15] based on the original work of Jordan [8], developed for a half bandwidth $n_{hb} = 64$. (Neither code involves pivoting.) The significance of these results can be summarized as follows.

Half Bandwidth	Factorization			Solution		
	NB	Sparse	Overhead	NB	Sparse	Overhead
2	3.8/14	1.1/48	51%	6.2/8.1	1.5/34	35%
4	9.6/19	4.3/42	31%	11/8.1	4.2/21	32%
8	23/28	14/46	11%	20/8.1	11/15	20%
16	52/45	34/68	3.1%	36/8.1	24/12	11%
32	88/210	62/296	.6%	65/18	42/28	4.7%
64	117/1260	91/1622	.1%	93/49	56/81	1.4%

Table 2. [Execution rates (MFLOPS)]/[time(kiloclocks)]
of narrow band (NB)- and sparse solvers. Sixty
four equations were solved for except for half
bandwidths of 32 and 64, where 128 and 256
equations were solved, respectively.

Square block size	Factor- ization Kernel M_f/T_f	Substi- tution Kernel M_s/T_s	Multipli- cation Kernel M_m/T_m
1	1.1/.073	.5/.15	2.7/.11
2	2.5/.20	3.0/.39	7.6/.16
3	6.9/.28	5.7/.63	17/.32
4	8.5/.42	12/.76	26/.40
6	18/.69	21/1.5	43/.78
8	23/1.1	30/2.6	60/1.4
12	45/2.0	46/5.8	84/3.3
16	60/3.5	58/11	98/6.7
32	94/18	89/58	124/42
64	123/112	117/358	141/298

Table 3 [Execution rate (MFLOPS)]/
[execution time (kiloclocks)]
of three numeric kernels. All
matrices are assumed square.

Block size	MFLOPS	η
1	.33	.35
2	1.9	.55
3	5.0	.66
4	10.	.72 (1.07)
6	21.	.83
8	32.	.90 (1.3)
12	54.	.95
16	69.	.97
32	102.	.995
64	126.	.999

Table 4. Estimated performance of
general block sparse solver
on the CRAY-1 for a large
system of block tridiagonal
equations.

Block size	MFLOPS range
1	.33 - 1.05
2	1.9 - 3.7
3	5.0 - 11.
4	10. - 18.
6	21. - 35.
8	32. - 53.
12	54. - 80.
16	69. - 96.
32	102. - 123.
64	126. - 141.

Table 5. Bounds on execution rates

- (1) The block sizes are equal to the half bandwidth for $n_{hb} \leq 64$. The resultant short vectors in the kernel processing inhibit performance more than the non-kernel overhead estimated in the table (see next section).
- (2) For $n_{hb} = 64$ the execution rate of the sparse solver is $\approx 25\%$ less than that of the narrow band code. This can be traced to relatively inefficient substitution codes included in the partitioned banded factorization and to the shortened vector lengths involved in the processing of the banded block models.
- (3) For large bandwidths greater than 64, the asymptotic speed is determined by the multiplication of 64×64 dense matrices, shown in Table 3 to be in the range of 141 MFLOPS.

An additional timing comparison was made between the sparse band solver and a sparse block tridiagonal solver. The latter can be used to solve a band matrix at a cost of additional arithmetic operations and storage, but with longer vectors. With block sizes equal to the matrix half bandwidth, the tridiagonal computation time was found to be approximately 36% larger than the solution time using the banded model of Figure 8, for $n_{hb} = 64$.

D. Other Timing Considerations

1. Efficiency

An interesting aspect of the general solver is that significant effort can be justified in development of the numeric kernels. In [11], for example, it is shown that $\approx 50\%$ speedup can be achieved in these kernels for small matrices (without penalty for large matrices) by avoiding vector chaining--through which the CRAY-1 normally achieves concurrency--and utilizing the vector registers instead as a dual cache memory which seeks to keep the floating point pipelines continuously busy. One can then view the speedup of these unconventional assembly language kernels as an asset which compensates for the overhead of list processing, the price for generality.

Define the efficiency η as the fraction of solution time to process the numeric kernels. Let all other time--subroutine linkage to load and unload backing (B and T) registers, list processing, instruction fetches, etc.-- be reviewed as overhead. The efficiencies associated with the factorization of the block tridiagonal matrices are given in Table 4.

An "adjusted efficiency" can then be obtained by multiplying the efficiencies of Table 4 by the kernel speedups. The product is the speedup of the general code over a code written for a block tridiagonal matrix with conventional assembly language kernels. These speedups are shown in parentheses in Table 4, for block sizes of four and eight, and range from 1.07 to 1.3.

From recent comparisons of Fortran and conventional assembly language codes [12], it is clear that speed comparisons of specialized Fortran-coded solvers and the general solver would result in an even greater advantage for the general solver. Thus, a user faced with programming a Fortran solver for a specific block structure would be well advised to consider use of this general solver requiring only a block descriptor input.

2. Bounds on Execution Rate

An interesting result of detailed timing study of the kernels in Table 3 is that, from an observation of the execution rate of the three kernels, it is possible to produce sparsity structures that yield the minimum and maximum execution rates for a particular block size. These rates therefore bound the execution rates for all sparse matrices which have a constant block size, or, depending on the bound being lower or upper, all sparse matrices having a known minimum or maximum block size, respectively.

From Table 3, the multiplication kernel consistently has a higher rate than either the factorization or substitution kernels, due to the absence of division and the larger number of computations per operand/result--thus reducing data flow. A symmetric structure with a large

number of off-diagonal blocks exercises this high speed kernel most often and should thus achieve the highest execution rate. By adding the multiplication loop overhead time to the multiplication kernels time of Table 3, the upper bound of execution rates of Table 5 is obtained.

The lowest rates result from the blocked tridiagonal matrix, among all structurally symmetric matrices, since only one high-speed multiplication is involved for every slower factorization and substitution step. Therefore, the execution rates from Table 3 become the lower bounds shown in Table 5.

Table 5 shows that execution rates vary over a rather narrow range ($\leq 2:1$), making these bounds quite useful when block size characteristics are known apriori.

E. Precise Timing Simulation

Timing being the critical issue in deciding on the adoption of the software, a more precise timing capability has been developed. Particular instances of the usefulness of this software are:

- (a) for a block sparsity structure other than a banded or tridiagonal pattern--for which timings has already been given--and when the block sizes are widely-varying so that Table 5 is not applicable;
- (b) when it is desirable to study the influence of alternate blocking strategies (perhaps resulting from alternate equation formulation methods) on solution time;
- (c) as a standard by which alternate codings (Fortran, special assembly coding) or alternate algorithms can be judged.
- (d) to estimate the solution time of very large systems of equations (≥ 10000 equations).

For these purposes, a Fortran block level timing simulator has been provided to run on a user's local machine, which may be more convenient and less expensive than a CRAY-1. (This should not be confused with the CRAY-1 clock-level simulator used to develop this code). When used in conjunction with a Fortran version of the sparsity software (also provided), a complete numerical and timing simulation of the solution can be carried out on any processor. In Table 6, the accuracy of the simulator is demonstrated

by comparing simulated timings of a full matrix solution with those produced from a clock-level CRAY-1 timing simulator [10]. The latter is normally accurate to within 1% of timings for a CRAY-1 with a million word (16 bank) memory.

The bandedge blocks defined in Figure 8 are timed as if they were full blocks. Although less computation is performed in operations involving these blocks, kernels associated with the processing execute at lower MFLOPS rates than the full block kernels.

Maximum Block Size	CRAY-1 Time (kiloclocks)	Simulated Time (kiloclocks)	Error (%)	Simu- lated Overhead (%)	Simu- lated MFLOPS
2-fac.	39.6	39.1	-1.2	38.2	5.35
-sub.	11.9	12.4	4.2	35.3	3.20
4-fac.	15.4	14.2	-7.8	26.1	14.6
-sub.	5.28	5.29	.2	32.6	7.50
8-fac.	7.68	7.23	-5.8	13.7	28.9
-sub.	2.89	3.01	4.2	25.5	13.2
16-fac.	3.83	3.80	-.8	8.5	55.0
-sub.	1.85	1.85	0	21.7	21.4

Table 6. Simulated timings for partitioned solution of 16x16 matrix, as function of block size. CRAY-1 time was obtained from a clock level simulator.

IV. PROGRAM DESCRIPTION AND USER INFORMATION

A. Introduction

The package of codes which perform the block oriented sparse solution (BOSS) are organized as follows:

- (a) BOSS/C. This is a collection of the CRAY-1 assembly codes which perform the numeric block solution from the symbolic and numeric block descriptors.
- (b) BOSS/S. This IBM standard Fortran package contains (1) Fortran versions of the above CAL codes, (2) a driver to illustrate use of the solver, (3) preprocessor routines that automatically block full, banded, and block tridiagonal matrices, and (4) a block-level timing simulator to estimate CRAY-1 timings and execution rates. These are intended for familiarization and debugging on a machine other than the CRAY-1.

B. A General Driver Program

1. High Level Descriptors

In Table 7 a compact Fortran driver program is displayed to illustrate use of all the major options of the software except BOSS/G. By choice of NLIB, the driver will either (a) allow user definition of the block structure (NLIB=1), or (b) produce block descriptors for full, block tridiagonal, or banded systems (NLIB=2,3, and 4, respectively). The user-supplied parameters for these cases are

```

C**** GENERAL DRIVER PROGRAM FOR BLOCK SPARSE SOLVER
      IMPLICIT REAL*8(A - H,O - Z)
C***  A IS PACKED MATRIX STORAGE; B IS RIGHT HAND SIDE
C      NBL = # OF DIAGONAL BLOCKS
C      NIRC = # * TOTAL NUMBER OF BLOCKS
C      DIMENSIONS ARE IC(NBL),JR(NBL),ID(NBL+1),IRC(NIRC)
      DIMENSION A(2000), B(300)
      DIMENSION IC(100), JR(100), ID(101), IRC(1000)
      READ (5,10) NLIB
10  FORMAT (12I5)
20  GO TO (30, 40, 50, 60), NLIB
C***  READ IN BLOCK DESCRIPTORS (IRC) AND POINTERS INTO IRC
C      THAT POINT INTO BEGINNING OF EACH DBCS (IC) AND EACH DBRS (JR)
30  READ (5,10) NBL, NIRC
      READ (5,10) (IC(J),J=1,NBL)
      READ (5,10) (JR(J),J=1,NBL)
      READ (5,10) (IRC(J),J=1,NIRC)
      NBP1 = NBL + 1
      READ (5,10) (ID(J),J=1,NBP1)
C***  EXPAND CONVERTS FROM 4 DESCRIPTORS/BLOCK TO 6 DES./BLOCK
      CALL EXPAND(NBL, NIRC, IC, JR, IRC, ID, A)
      GO TO 70
C***  GENERATE FULL MATRIX DESCRIPTOR LIST FROM
C      NMAT (MATRIX SIZE), NDIM (ROW DIMENSION OF FULL MATRIX ARRAY),
C      AND NPB (MAX SIZE OF BLOCK - USUALLY 64)
40  READ (5,10) NMAT, NDIM, NPB
      CALL FULL(IC, JR, IRC, ID, NBL, NIRC, NPB, NMAT, NDIM)
      GO TO 70
C***  GENERATE BLOCK TRIDIAGONAL DESRIPTOR LIST FROM
C      NBL(# OF DIAG BLOCKS), NPB (SIZE OF BLOCKS) AND
C      NSA, NSB, AND NSC, THE STARTING ADDRESSES IN NUMERIC
C      ARRAY OF DIAG., UPPER DIAG., AND LOWER DIAG. STRIPES
50  READ (5,10) NBL, NPB, NSA, NSB, NSC
      CALL TRIEL(IC, JR, IRC, ID, NBL, NIRC, NPB, NSA, NSB, NSC)
      GO TO 70
C***  GENERATE BAND MATRIX DESCRIPTOR LIST FROM
C      NBW (HALF BANDWIDTH), NMAT (MATRIX SIZE),
C      NDIM (ROW DIMENSION OF BAND MATRIX ARRAY),AND
C      NPB (MAX SIZE OF BLOCK - USUALLY 64)
60  READ (5,10) NBW, NMAT, NDIM, NPB
      CALL BAND(IC, JR, IRC, ID, NBL, NIRC, NMAT, NBW, NDIM, NPB)
70  CONTINUE
C***  NB = # OF RIGHT HAND SIDES (RHS);
C      NDIMB = ROW DIMENSION OF RHS ARRAY B
C      DIMENSION OF B IS NB*NDIMB
      NB = 1
      NDIMB = 300
C***  FORMULATE EQUATIONS IN SUBROUTINE FORM
80  CALL FORM(A, B, IC, JR, IRC, ID, NBL, NB, NDIMB)
C***  TIMER PERFORMS SIMULATED TIMINGS
      CALL TIMER(ID, IRC, IC, JR, NBL, NB)
C***  FAC PERFORMS NUMERIC FACTORIZATION
C      SOL PERFORMS FBR. & BACK SUBSTITUTION
90  CALL FAC(A, NBL, IC, JR, IRC, ID)
      CALL SOL(A, NBL, B, NB, NDIMB, IC, JR, IRC, ID)
C***  ID(NBL+1) = # OF EQUATIONS + 1
      NBP1 = NBL + 1
      NBB = ID(NBP1) - 1
      WRITE (6,100) (B(J),J=1,NBB)
100  FORMAT (5E13.5)
      STOP
      END

```

Table 7. General driver program

shown below; here A is the matrix array and B the right hand side array.

(a) Full matrix

NMAT--the number of equations

NDIM--the row dimension of the A array, to allow
 $NDIM \geq NMAT$.

NPB--the maximum number of elements per diagonal
block, usually 64 for the CRAY-1.

(b) Block tridiagonal matrix stored in three strips of
diagonal, upper diagonal, and lower diagonal blocks,
all column ordered (see Figure 7).

NBL--the number of diagonal blocks.

NPB--size of blocks.

NSA--starting address in A of the diagonal blocks
(see Figure 7).

NSB--starting address in A of the upper diagonal
blocks.

NSC--starting address in A of the lower diagonal
blocks.

(c) Band matrix, stored in compressed form so that the (1,1)
position of the matrix is located in A(NBW+1,1), the
(2,2) position in A(NBW+1,2), etc.

NBW--the half bandwidth

NMAT--the number of equations

NDIM--the row dimension of the A array, to allow
 $NDIM \geq 2NBW + 1$

NPB--the maximum number of elements per diagonal
block, usually 64.

The subroutines that perform the blockings return arrays IC, JR, IRC,
and ID and variables NBL, NIRC, and NMAT, if they are not already
defined prior to the subroutine call.

The coding of the forward and back substitution favors simul-
taneous substitutions of multiple right hand sides rather than
multiple calls to subroutine SOL. Indeed, two right hand sides
can be substituted in only slightly more time than a single right
hand side. Two parameters must be defined regardless of the number
of substitutions.

NB--the number of right hand sides

NDIMB--the row dimension of the B array, or equivalently, the address increment between the columns of B.

Computations will be performed on an even number of columns of B, regardless of the value of NB; however, results of only NB columns will be stored back to main memory. Sufficient storage should be devoted to B so that the last even column does not contain instructions. Thus, a single right hand side should be dimensioned at least twice the value of NDIMB.

2. Equation Formulation

Subroutine FORM is intended to be supplied by the user to formulate the equations in arrays A and B. In such cases, the only arguments required in FORM are A and B; these arrays are filled by the user according to the structure of the numeric storage assumed by the blocking (e.g., full, compressed banded). The FORM subroutine supplied with the package was developed to check the solution software, and automatically formulates a set of equations from the block descriptors so that a numeric solution $B(r)=r$ is obtained.

3. A Fortran Simulator

Subroutine TIMER produces simulated timings for both the factorization and substitution steps. This includes estimates of the fraction of time devoted to overhead, i.e., time not included in numeric kernels. An example output is shown in Figure 10(c).

Fortran versions of subroutines FAC and SOL are also contained in BOSS/S; these general subroutines have the same calling arguments as the CRAY-1 assembly-coded version. This allows development of the blocking descriptors and the user-supplied equation formulation subprogram (FORM) on a machine other than the CRAY-1.

C. Generation of Block Descriptors

For a general block structure, the user must supply (1) an array IRC of block descriptors, (2) arrays IC and JR that point to the beginning of column and row strips (respectively) in IRC, and (3) array ID, which denotes the first row number of the diagonal blocks.

These are summarized in Table 8.

An abbreviated descriptor list without b3 and b6 can be supplied when only full blocks are anticipated and when the user is well acquainted with the general requirement the blocks must be contained within row or column strips. The subroutine EXPAND will then add b3 and b6 (=0) to the list, assuming that additional storage has been allocated to IRC.

An example structurally unsymmetric matrix example is illustrated in Figure 10. Each block is column ordered and all blocks are stored in the numbered ordering shown. The corresponding input data file is given in Table 10(b). The Fortran simulator produces the timing estimate of Table 10(c). The execution rates are shown to be less than 7% the maximum execution rate of the machine; the overhead attributable to nonkernel processing--i.e., the price of generality--is approximately 30% of the total. Thus, most of the relatively low performance is due to the overwhelming number of 3x3 and similarly-sized blocks.

Descriptors for blocks in Ith diagonal column and row strips

- IC(I): Address in IRC of first descriptor for Ith diagonal block.
- JR(I): address in IRC of first descriptor for first block to right of Ith diagonal block.
- ID(I): Row (column) number of first element of Ith diagonal block; ID(NBL+1) must be applied equal to the number of equations +1.

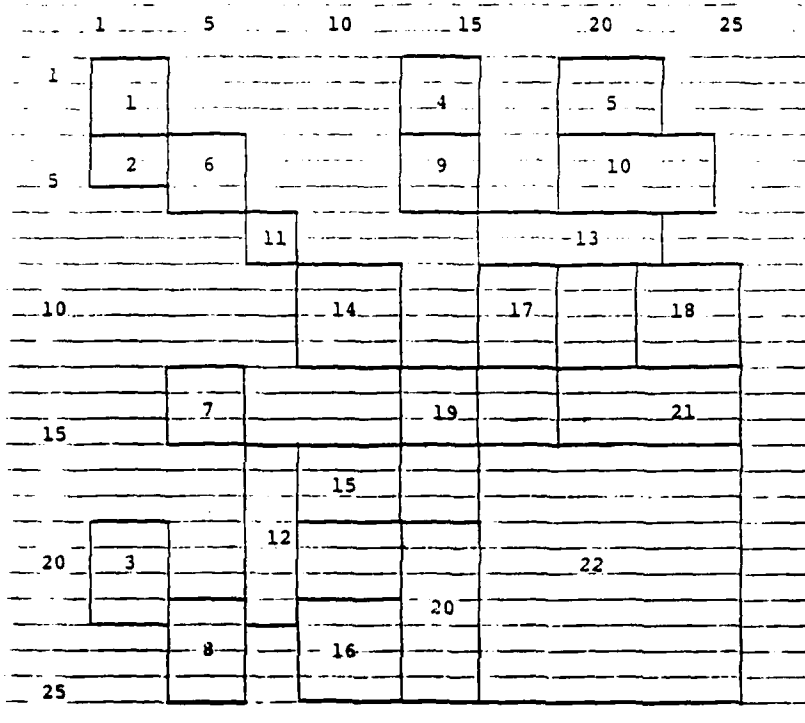
For block in column strip ($II = 6*(I-1)$)

- b1. IRC(II+1): row # of (1,1) position of block.
- b2. IRC(II+2): # of rows in block.
- b3. IRC(II+3): # of row strip containing block.
- b4. IRC(II+4): location in numeric array A of (1,1) position of block.
- b5. IRC(II+5): address increment in numeric array A between (1,1), and (1,2) positions of block.
- b6. IRC(II+6): = 0 for full block; = 1 for bandedge block.

For block in row strip: ($II = 6*(I-1)$)

- b1. IRC(II+1): column # of (1,1) position of block.
- b2. IRC(II+2): # of columns in block.
- b3. IRC(II+3): # of column strip containing block.
- b4. IRC(II+4): location in numeric array A of (1,1) position of block.
- b5. IRC(II+5): address increment in numeric array A between (1,1) and (1,2) positions of block.
- b6. IRC(II+6): = 0 for full block; = 1 for bandedge block.

Table 8. User-defined blocking arrays for NLIB = 1.



(a) Structure

1												
6	88											
1	21	41	53	73	85							
13	33	49	65	81	89							
1	3	1	3	4	2	10	2	19	4	16	4	
13	3	28	3	19	4	37	3	4	3	49	3	
13	3	58	3	22	4	67	4	13	3	79	3	
19	6	88	3	7	2	106	2	16	7	110	7	
16	7	124	2	9	4	138	4	16	3	154	3	
22	4	166	4	16	3	182	4	22	4	194	4	
13	3	210	3	19	7	219	7	19	7	240	3	
16	10	261	10									
1	4	7	9	13	16	26						

(b) Input data (4 descriptors/block)

	TIME MILLOCLKS	SUB. LINK. OVERHD	OTHER OVERHD	MFLOPS
FAC	20.424	0.9%	30.1%	10.90
SOL	8.149	2.1%	30.3%	6.82

(c) Simulator output

Figure 10. Solution of a block sparse matrix.

V. Conclusions

This report illustrates the specialized algorithm and coding effort associated with achieving optimal performance from a memory-hierarchical vector processor such as the CRAY-1. Indeed, for sparse matrices without a pronounced blocked structure, it is still often possible to achieve high performance by utilization of other matrix characteristics [16]; however, a vastly different vectorization process is necessary resulting in a package that solves general simultaneous sparse systems. Combining these solvers then produces a polyalgorithm which is of even more general application.

For large systems one can argue that provisions for pivoting and I/O partitioning should be included in a general package, and that "handles" should be provided for interfacing with iterative methods for use in partial elimination methods.

In summary, the development of a "general" high performance solver on a complicated scientific architecture is a demanding task. In this context, the block-oriented solver of this report can be viewed as a demonstration code that efficiently solves a broad class of sparse problems, but is necessarily more restricted than familiar scalar general sparse solvers intended for smaller problems. Other general sparsity software is being studied for other classes of sparsity structures [17]

References

- [1]. W. F. Tinney, J. W. Walker, "Direct Solutions of Sparse Network Equations by Optimally Ordered Triangular Factorization," Proc. IEEE, vol. 55, November, 1967, pp. 1801-1809.
- [2]. R. P. Tewarson, "Solution of a System of Simultaneous Linear Equations with a Sparse Coefficient Matrix by Elimination Methods," BIT, vol. 7, 1967, pp. 226-239.
- [3]. F. G. Gustavson, "Some Basic Techniques for Solving Sparse Systems of Linear Equations," Sparse Matrices and Their Applications, D. J. Rose and R. W. Willoughby Eds., Plenum Press, 1972, pp. 41-52.
- [4]. G. von Fuchs, J. R. Roy, and E. Schrem, "Hypermatrix Solution of Large Sets of Symmetric Positive Definite Linear Equations," Comput. Math. Appl. Mech. Engring., vol 1, 1972, pp. 197-216.
- [5]. D. A. Calahan, "Vectorized Sparse Elimination," Proc. SCIE Meeting on Impact of Advanced Systems on Scientific Computation, Livermore, CA, Spt. 12-13, 1979.
- [6]. D. A. Calahan, Computer-Aided Network Design, McGraw-Hill, 1972.
- [7]. D. Boley, B. L. Buzbee, and S. V. Parter, "On Block Relaxation Techniques," Report #318, Computer Sciences Dept., University of Wisconsin, Madison, June, 1978.
- [8]. R. Fong, T. L. Jordan, Some Linear Algebraic Algorithms and Their Performance on the CRAY-1, Report LA-6774, Los Alamos Scientific Laboratory, June, 1977.
- [9]. A. George, "Numerical Experiments Using Dissection Methods to Solve n by n Grid problems," SIAM J. Numer. Anal., vol. 14, April, 1977, pp. 161-179.
- [10]. D. A. Orbits, A CRAY-1 Timing Simulator, Report #118, Systems Engineering Laboratory, University of Michigan, September, 1978.
- [11]. W. G. Ames, et al, Sparse Matrix and Other High Performance Algorithms for the CRAY-1, Report #124, Systems Engineering Laboratory, University of Michigan, January, 1979.
- [12]. J. J. Dongarra, Linpack Working Note #11: Some Linpack Timings on the CRAY-1, Report #LA-7389-M, Los Alamos Scientific Laboratory, August, 1978.
- [13]. D. A. Calahan, et al, "A Collection of Equation-Solving Codes for the CRAY-1," Report #134, Systems Engineering Laboratory, University of Michigan, Ann Arbor, August, 1979.

- [14]. D. A. Calahan, "Performance of Linear Algebra Codes on the CRAY-1," Proc.. 5th Symp. on Reservoir Simulation, Soc. Pet. Engrs., Denver, Jan. 31, 1979, pp. 7-15.
- [15]. D. A. Calahan, "A Collection of Equation-Solving Codes for the CRAY-1," Report #133, Systems Engineering Laboratory, Univ. of Michigan, August 1, 1979.
- [16]. D. A. Calahan, "Multi-level Vectorized Sparse Solution of LSI Circuits," Proc. 1980 IEEE Conf. on Circuits and Computers, Rye, N.Y., pp. 976-989.
- [17]. D. A. Calahan, "Performance of Linear Algebra Codes on the CRAY-1," Journal Soc. Pet. Engrs., (to be published, Feb., 1981).